CS2106: Introduction to Operating Systems

Lab Assignment 1 (A1)
Advanced C Programming and Shell Scripting

Important:

The deadline of submission through LumiNUS: Sat, 5 Sep, 2pm
The total weightage is 7%:

- Exercise 1: 2 % [Lab demo exercise]
- Exercise 2: 2 %
- Exercise 3: 0 %
- Exercise 4: 2 %
- Exercise 5: 1 %
- Exercise 6: 0 %

You must ensure the exercises work properly on the SoC Compute Cluster, h*ostnames: xcne0 - xcne7, Ubuntu 20.04, x86_64, gcc 9.3.0*.

Section 1. General Information

Here are some simple guidelines that will come in handy for all future labs.

1.1. Lab Assignment Duration & Lab Demonstration

Each lab assignment spans about **two weeks** and consists of multiple exercises. One of the exercises is chosen to be the "lab demo exercise" which you need to demonstrate to your lab TA. For example, in this lab, you need to demo exercise 1. The demonstration serves as a way to “kick start” your effort. You are **strongly encouraged to** finish the demo exercise before coming to the lab.

The remaining lab exercises are usually quite intensive. Do not expect to finish the exercise during the allocated lab session. The main purpose of the lab session is to demo your exercise and clarify doubts with the lab TAs.

1.2. Lab Setup

Since all classes run online, we do not have a physical lab where you can go and sit down to complete your lab assignments. This semester:

- You will use the SoC Compute Cluster to test your assignments.
- Additionally, we provide a virtual machine image with Ubuntu 20.04 installation that can be used with VirtualBox on your personal computer.
- Alternatively, you might install Ubuntu 20.04 natively on your personal computer.

You may use any of the three methods above to develop your assignments. However, take note that:

Your submissions will be tested on one of the following machines from SoC Compute Cluster: hostnames (nodes) xcne0 - xcne7.

To read details about the nodes on the SoC Compute Cluster check:
https://dochub.comp.nus.edu.sg/cf/guides/compute-cluster/hardware

Nodes xcne6 and xcne7 have been reserved for CS2106 for the entire semester. However, you may use any available node xcne0 - xcne7 to test your assignments before submissions.

The software configuration for these nodes follows:

```
Ubuntu 20.04
gcc version 9.3.0
GNU bash, version 5.0.16(1)-release
```

You must use your SoC account to use these nodes. If you do not have an SoC account, you can retrieve or create one here:
https://mysoc.nus.edu.sg/~newacct/

To use these nodes, you must enable SoC Compute Cluster from your MySoC Account page (https://mysoc.nus.edu.sg/~myacct/services.cgi).

To remotely connect to any of these nodes, simply SSH to them from the SoC network using your SoC account details (used to connect to `sunfire`) as follows:

- First connect to `sunfire` using your SoC credentials
- After successful connection, from `sunfire` console, run the command:
  `$ ssh xcne6`

This command connects to the `xcne6` node of the Compute Cluster.

You should not use these nodes on the SoC Compute Cluster to store your assignment code.

You might setup a **private git repository (on GitHub)** for your development and retrieve the latest version of your code using git. Alternatively, you can use `scp` to transfer files in and out of the compute cluster nodes.

1.3. Setting up the exercises

For every lab, we will release two files in LumiNUS "Labs" files:

- **labX.pdf**: A document to describe the lab question, including the specification and the expected output for all the exercises.
- **labX.tar.gz**: An archive for setting up the directories and skeleton files given for the lab.

For unpacking the archive:

1. Copy or download the archive **lab*X*.tar.gz** into your account.
2. Enter the following command in the terminal (console):
   tar -zxvf lab*X*.tar.gz
   OR gunzip –c lab*X*.tar.gz | tar xvf –
   Remember to replace the ***X*** with the actual lab number.
3. The above command should setup the files in the following structure:

ex1/		subdirectory for exercise 1
	ex1.c	skeleton file for exercise 1
	testY.in	sample test inputs, Y= 1, 2, 3, ...
	testZ.out	sample outputs, Z = 1, 2, 3, …
ex2/		
	...	Similar to ex1
ex3/		Similar to ex1
...		

Section 2. Exercises in Lab 1

There are **six exercises** in this lab. The main motivation for this lab is to familiarize you with some:

- advanced aspects of C programming,
- compiling and running C programs in Linux, and
- using the shell in Linux.

As such, you will need to write a combination of C programs and shell scripts (shell commands) to achieve some simple tasks. The **techniques and shell commands** used in these exercises are quite commonly used in OS related topics, and they will help you in completing the next lab assignments.

2.1. Setup

Before we get started on the exercise, we want to get you familiarized with basic shell commands. Instead of downloading the zip folder from LumiNUS, we will download it using the terminal!

You can think of the terminal as an interactive program that allows you to type commands to complete some tasks without a graphical interface. You will be in a terminal environment once you `ssh` into the nodes from SoC Compute Cluster, or open the terminal application on a Linux desktop. Instructions about how to access these nodes can be found in Section 1.2.

To navigate around in the terminal, you need to be familiar with basic commands such as:

- `pwd`: print current working directory
- `cd`: change directory
- `ls`: list files in current directory

You can also type `man <command>` on the terminal to view the user manual of the command and read more about it.

To start the lab, we first need you to fetch the zip files from a download link. You can use either `wget` or `curl` to do so. Use `man / help` to find out how to use `wget` or `curl`.

Download link:
https://www.comp.nus.edu.sg/~ccris/cs2106_ay2021s1/lab1.tar.gz

Once the files are downloaded, follow the instructions in Section 1.3 to unpack the files for lab1.

2.2. Exercise 1: Doubly Linked List in C [Lab Demo Exercise] (1% demo + 1% submission)

Exercise 1 requires you to implement in C some functionalities for a doubly linked list. Doubly linked lists are used in the implementation of `runqueue in` the Linux Kernel. This exercise allows you to become more familiar with C syntax and appreciate the challenge behind implementing different parts of the operating system.

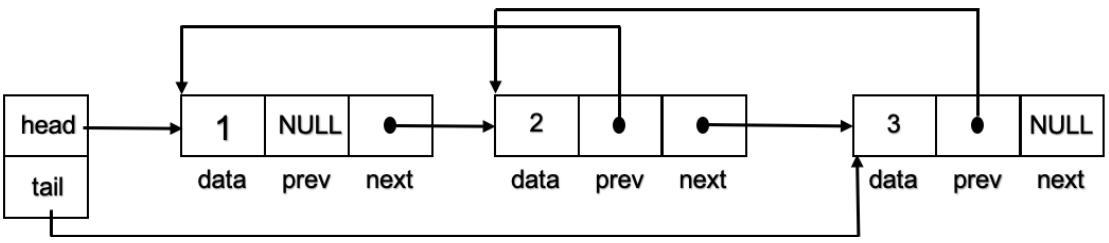
For this exercise, we represent a node in our doubly linked list as follows (in `node.h`):

```
typedef struct NODE
{
      struct NODE *prev;
      struct NODE *next;
      int data;
} node;
```

The list representation looks like this:

```
typedef struct
{
      node *head;
      node *tail;
} list;
```

Every node has a pointer to its prev (left) and next (right) node. The list points to the head (front) and tail (back). A pictorial representation follows:



You need to implement five functions shown below to work with the doubly linked list:

`void insert_node_from_head_at(list *lst, int index, int data)`
This function inserts a new node that contains **data** at **index** of **lst** counting from the head (index starts from 0). You should use **malloc** to allocate memory to create the new node. Assume that **index** is between 0 and length of the list, inclusive.

`void insert_node_from_tail_at(list *lst, int index, int data)`
This function inserts a new node that contains **data** at **index** of **lst** counting from the tail (index starts from 0). You should use **malloc** to allocate memory to create the new node. Assume that **index** is between 0 and length of the list, inclusive.

`void delete_node_from_head_at(list *lst, int index)`
This function deletes the node at **index** of **lst** counting from the head (index starts from 0). You should use **free** to delete memory allocated. Assume that **index** exists in the list.

`void delete_node_from_tail_at(list *lst, int index)`
This function deletes the node at **index** of **lst** counting from the tail (index starts from 0). You should use **free** to delete memory allocated. Assume that **index** exists in the list.

`void reset_list(list *lst)`
This function deletes all nodes in the list and resets head and tail to NULL.

For this exercise it is important to note that the list will start with having head and tail set to **NULL** (this is how we represent an empty list). The first node in our list will have its previous pointer set to **NULL** and the last node in our list will have its next pointer set to **NULL**.

The folder structure of **ex1** is as follows:

/ex1
node.h **(not to be modified)** ex1.c **(not to be modified)** node.c *.in / *.out (files used for testing) Makefile **(not to be modified)**

You should modify **node.c** for this exercise. Please take note that changes to any other file will be overwritten when we run the grading script.

After you have finished implementation, use the following command to compile your code:

```
$ gcc -std=c99 -Wall -Wextra node.c ex1.c -o ex1
```

This will produce the executable **ex1** by running `gcc` compiler with c99 standard, enabling warnings, and creating an output `ex1`.
You may use `-Werror` option in `gcc` to make all warnings into errors.

You can use the sample test case we have provided to test your code.

`$ ./ex1 < sample.in | diff sample.out -`
(Note the `-` at the end!)

The above bash command passes the **`sample.in`** input file into the test runner and compares the output against the expected. Details about how this command runs follow:

- The bash spawns two processes. The first process runs `ex1` and the second process runs `diff`.
- We used **input redirection (<)** to replace the standard input (stdin) with the file `sample.in` for the first process (running ex1)
- We have made use of a **pipe (|)** to pass the output from the first process (running the command **`./ex1 < sample.in`)** into the input of the second process running (running the command **`diff sample.out -`**). The – sign stands for the second input file being replaced with standard input (here, with the output produced by ex1).

Apart from **`sample.in`**, we have two more test cases to help verify your program. To get the demo exercise grade for this lab, you have to show your lab TA that you are familiar with basic bash syntax and have a working doubly linked list that works for all test cases.

Also take note that the runner frees up any memory it allocates and the list before it terminates the program. The only file that needs changes is **`node.c`**.

Refer below for an explanation of the sample input. The input file uses a specific numbering scheme to refer to the five functions defined above:

`sample.in`

```
1 0 1   //insert_node_from_head_at(lst, 0, 1)
0       //print_list(lst)
2 0 3   //insert_node_from_tail_at(lst, 0, 3)
0
1 1 2
0
1 0 100
0
2 0 200
0
3 0     //delete_node_from_head_at(lst, 0)
0
4 0     //delete_node_from_tail_at(lst, 0)
0
```

```
5       //reset_list(lst)
1 0 1000
0
```

sample.out

```
Forward: [ 1 ], Backwards: [ 1 ]
Forward: [ 1 3 ], Backwards: [ 3 1 ]
Forward: [ 1 2 3 ], Backwards: [ 3 2 1 ]
Forward: [ 100 1 2 3 ], Backwards: [ 3 2 1 100 ]
Forward: [ 100 1 2 3 200 ], Backwards: [ 200 3 2 1 100 ]
Forward: [ 1 2 3 200 ], Backwards: [ 200 3 2 1 ]
Forward: [ 1 2 3 ], Backwards: [ 3 2 1 ]
Forward: [ 1000 ], Backwards: [ 1000 ]
```

We defined specific macros for each of the functions:

```
#define PRINT_LIST 0
#define INSERT_FROM_HEAD_AT 1
#define INSERT_FROM_TAIL_AT 2
#define DELETE_FROM_HEAD_AT 3
#define DELETE_FROM_TAIL_AT 4
#define RESET_LIST 5
```

The function **print_list** has already been written for you within the runner ex1.c. Feel free to look at the **ex1.c** to understand how the runner works. Understanding how the runner works will help greatly when doing the next exercise.

2.3. Exercise 2: Function Pointers – 2%

This exercise extends the functionalities of the doubly linked list implementation by applying several operations on its nodes. These operations are added by making use of function pointers in C.

Function Pointer Overview

You can refer to this link for more information on function pointers. Unlike normal pointer, which points to memory location for **data storage,** a function pointer **points to a piece of code (function)**. By dereferencing a function pointer, we **invoke the function** that is referred by that pointer. This technique is commonly used in **system call / interrupt handlers**.

In C, it is possible to define a **function pointer** to refer to a function. For example:

```
void (*fptr) ( int );
```

To understand this declaration, imagine if you replace **`(*fptr)`** as **`F`**, then you have:

```
void F( int );
```

So, **`F`** is "a function that takes an integer as input, and return nothing (void)". Now, since **`(*fptr)`** is **`F`**, **`fptr`** is "**a pointer to** a function that takes an integer as input, and return nothing (void)".

Let's use the function pointers to define and use a group of functions that can **map** different operations to the doubly linked list from exercise 1.

Exercise 2 is an extension of exercise 1. For this exercise, you must write the test runner `ex2.c` and `node.c`. The runner

- **reads the input file provided as a command line argument** (reading from a file can be done using any library. We recommend using `stdio (fopen, fclose, fread).` Make sure that you gracefully handle an invalid file name.
  and
- **applies the operations listed in the input file** on the doubly linked list.
  The macros corresponding to the functions that can be applied on the list are:

```
#define SUM_LIST 0
#define INSERT_FROM_HEAD_AT 1
#define INSERT_FROM_TAIL_AT 2
#define DELETE_FROM_HEAD_AT 3
#define DELETE_FROM_TAIL_AT 4
#define RESET_LIST 5
#define MAP 6
```

INSERT and DELETE operations are similar with exercise 1 (no changes are needed). We have added **MAP** and replaced **PRINT_LIST** with **SUM_LIST**.

SUM_LIST function definition is provided in `node.h`. This function sums the data of all nodes in the list and prints out (at standard output) the sum.

In addition to the functionalities from exercise 1, we define a `map` function:

`void map(list *lst, int (*func) (int))`
This function updates **lst** by applying **func** to the **data** element of every node

The **map** function (`MAP`) uses function pointers. You need to implement this function by applying the function `func` on each element of the doubly linked list.

MAP can be used to apply five operations on the list. Indices for these operations are given below:

0	**add_one**
1	**add_two**
2	**multiply_five**
3	**square**
4	**cube**

The implementation for these functions is provided in file `functions.c`. The runner `ex2.c` simply calls the right function based on the index given in the input file.

To apply the five `MAP` operations, you should create an array of function pointers with indices 0 to 4. Use the index from the input file to call the corresponding map function. This array of function pointers should be:

- named `func_list;`
- declared in `function_pointers.h` file
- initialized using a function named `update_functions` (defined in `function_pointers.h` and implemented in `function_pointers.c`). `update_functions` is called in the main function of `ex2.c` (please do not modify this call).

The runner should be easily extensible to allow for new operations for the `map` function without changing the implementation (in `ex2.c` and `node.c`). Adding (removing) a new `MAP` operation should be done by modifying only the array of function pointers in `function_pointers.c`.

For this exercise the file structure is as follows:

/ex2
Makefile **(not to be modified)**
node.h **(not to be modified)**
ex2.c
node.c
function_pointers.h
function_pointers.c
functions.c **(not to be modified)**
functions.h **(not to be modified)**
*.in / *.out (files used for testing)

As explained earlier, we provide the functions used by **MAP** in **functions.c** and **functions.h**. These files should not be modified.

Use the Makefile provided to compile your code:
```
$ make
```

Command make uses the instructions found in the Makefile to compile your code. You can also run **make clean** to clean up the executable.

Test your ex2 as follows:

$ ./ex2 sample.in > res.out
$ diff res.out sample.out (to compare your output with the given output)

Apart from the sample, we have provided two more test cases for testing. Do take note that getting the right answer for these files will not guarantee full marks for this exercise (see exercise 3). Please test your code rigorously on your own. Other test cases will be used during grading.

Your runner should also free any memory it allocates and reset the list before the program terminates.

A sample input and output are shown below:

sample.in:
```
1 0 1  // first three same as ex1
2 0 3
1 1 2
0      // sums list and prints it
6 0    // runs map on list with add_one function
0
6 3    // runs map on list with square function
0
6 4    // runs map on list with cube function
0
```

sample.out:
6 9 29 4889

When we pass **sample.in** to our program, it should output at standard output (console) the result of any **SUM_LIST** instruction found. This is why the **sample.out** file has 4 numbers (4 **SUM_LIST** instructions in our input file).

Some things to take note of for this exercise:

- input will always be valid (there is no need to do input validation on runner)
- the sum of list will be smaller than 2^63-1 for any test cases we use (notice we use **long** data type for the function definition)

2.4. Exercise 3: Checking for Memory Errors – 0%

In this exercise, we introduce **valgrind**, a tool commonly used to identify and fix any kind of memory errors (memory leak detection / out of bound array access etc.). You might use **valgrind** for future lab assignments.

We want you to try using **valgrind** on the executable from **ex1** and **ex2**. You used **malloc** or **free** a couple of times in the code and there is a possibility of memory leaks occurring if any resource obtained dynamically is not freed.

To use valgrind, **cd** into **ex2** directory and run the below command:
valgrind ./ex2 sample.in > res.out

You should see output similar to this:

```
==3368== Memcheck, a memory error detector
==3368== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==3368== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==3368== Command: ./ex2 sample.in
==3368==
==3368==
==3368== HEAP SUMMARY:
==3368==     in use at exit: 0 bytes in 0 blocks
==3368==   total heap usage: 7 allocs, 7 frees, 8,824 bytes allocated
==3368==
==3368== All heap blocks were freed -- no leaks are possible
==3368==
==3368== For counts of detected and suppressed errors, rerun with: -v
==3368== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

As you can see, **valgrind** has done the hard work of checking for any potential memory leaks for us! Suppose our **ex2** had some memory leaks. **valgrind** detects memory problems and shows how to rerun to see additional details. When running on the sample input, you might see output as follows:

```
==3388== Memcheck, a memory error detector
==3388== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==3388== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==3388== Command: ./ex2 sample.in
==3388==
==3388==
==3388== HEAP SUMMARY:
==3388==     in use at exit: 8 bytes in 1 blocks
==3388==   total heap usage: 7 allocs, 6 frees, 8,824 bytes allocated
==3388==
==3388== LEAK SUMMARY:
==3388==    definitely lost: 8 bytes in 1 blocks
==3388==    indirectly lost: 0 bytes in 0 blocks
==3388==      possibly lost: 0 bytes in 0 blocks
==3388==    still reachable: 0 bytes in 0 blocks
==3388==         suppressed: 0 bytes in 0 blocks
==3388== Rerun with --leak-check=full to see details of leaked memory
==3388==
==3388== For counts of detected and suppressed errors, rerun with: -v
==3388== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

This exercise is not graded but take note that we will deduct marks if there are memory errors in either **ex1** or **ex2** (even if the expected output is correct).

2.5. Exercise 4: Shell scripting to find out more about our system – 2%

In the next two exercises, we will write some shell scripts! A shell script is a list of commands designed to be run by the Linux shell. Earlier when we talked about **cd** and **pwd**, commands you can use in the terminal (shell). You can think of a shell script as running a sequence of these commands. For this lab, we will be focusing on the **bash** shell.

Exercise 4 requires you to write a simple shell script to learn more about your operating system and system. Use the man pages and online search to find out about **bash**.

A simple **bash** shell script that prints the current working directory is as follows:

check_dir.sh

```
#!/bin/bash
dir=$(pwd)
echo Current directory: $dir
```

The first line of the script is known as an *interpreter directive*, and it starts with the magic sequence **#!** known as a shebang. The directive is parsed by the kernel and instructs the kernel to run the script using the given program, in this case **/bin/bash**. This directive can also be used for other shells (e.g. **zsh**, **fish**) and also for other scripting languages (e.g. Python).

Following this, we just have a variable that takes in the return of **pwd** command and we print it out using **echo**.

To run the above script, use the following commands:

```
$ chmod +x ./check_dir.sh
$ ./check_dir.sh
```

The first command helps to change the file permissions to make the script executable. The second command runs the script. You should be able to see your current directory being printed onto the screen.

The output of the script you are supposed to write for this exercise should look as follows:

Expected Output

```
Hostname: xcne7
Linux Kernel Version: 5.4.0-40-generic
Total Processes: 104
User Processes: 10
Memory Used (%): 6.10916
Swap Used (%): 0
```

The actual values differ based on the system you are using. We have provided in **ex4** folder a skeleton **bash** script **check_system.sh** that you can use to start work on this. You do not have to install anything else on your system to get the above information.

As a hint you should be looking into some of the following commands:

- **uname**
- **ps**
- **free**
- **awk** (to help computing the percentages)
- **pipe in shell (|)**

Do check the `man` pages to find out more about these commands. You can always run these commands in the terminal to test and copy them into the **bash** script once they give the output you desire.

2.6. Exercise 5: Know your `syscalls` – 1%

For this exercise, we will be writing another shell script to help us list the system calls done by a C program. A system call allows a user program to request services that are provided by the operating system. To find out what system calls are made by a program, we use a tool known as **strace**.

Within the **ex5** folder, you may find a sample C program **pid_checker.c** that prints its own process ID and its parent's process ID. We have also given a **bash** script skeleton **check_syscalls.sh** for you to update.

You need to complete the **bash** script to obtain a report on the system calls made by the program and the time spent in each call.

The expected output has the following format (values might differ, depending on the program that you are running):

Expected Output

```
Printing system call report
Process ID: 1745
Parent Process ID: 1745
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
  0.00    0.000000           0         1           read
  0.00    0.000000           0         2           write
  0.00    0.000000           0         2           open
  0.00    0.000000           0         2           close
  0.00    0.000000           0         3           fstat
  0.00    0.000000           0         7           mmap
  0.00    0.000000           0         4           mprotect
  0.00    0.000000           0         1           munmap
  0.00    0.000000           0         3           brk
  0.00    0.000000           0         3         3 access
  0.00    0.000000           0         1           getpid
  0.00    0.000000           0         1           execve
  0.00    0.000000           0         1           arch_prctl
------ ----------- ----------- --------- --------- ----------------
100.00    0.000000                    31         3 total
```

Again, reading the Linux manual for **strace** should allow you to do this exercise rather quickly.

2.7. Exercise 6: Check your archive before submission – 0%

Before you submit your lab assignment, run our check archive script named **check_zip.sh.**
The script checks the following:

a. The name or the archive you provide matches the naming convention mentioned in Section 3
b. Your zip file can be unarchived, and the folder structure follows the structure presented in Section 3
c. All files for each exercise with the required names are present
d. Each exercise can be compiled and/or executed.
e. The output your exercise produces using our sample input matches the expected output.

Once you have the zip file, you will be able to check it by doing:

```
$ chmod +x ./check_zip.sh
$ ./check_zip.sh E0123456.zip (replace with your zip file name)
```

During execution, the script prints if the checks have been successfully conducted, and which checks failed. Successfully passing checks a.-d. ensures that we can grade your assignment. **Points might be deducted if you fail these checks.**

Expected Successful Output

```
Checking zip file....
Unzipping file: E0123456.zip
Transferring necessary skeleton files
[…]
ex1: Success
ex2: Success
ex4: Success
ex5: Success
```

Section 3. Submission through LumiNUS

Zip the following files as E0123456.zip (**use your NUSNET id, NOT your student no A012…B, and use capital 'E' as prefix**):

Do **not** add any additional folder structure during zipping. The file structure should be:

E0123456.zip contains 4 folders, with the following content:

```
ex1/
      node.c
ex2/
      function_pointers.h
      function_pointers.c
      node.c
      ex2.c
ex4/
      check_system.sh
ex5/
      check_syscalls.sh
```

*The bolded names are folders.

Upload the zip file to the "Student Submission→Lab 1" folder on LumiNUS. Note the deadline for the submission is **5 Sep, 2pm.**

Please ensure that you follow the instructions carefully (output format, how to zip the files etc). Deviations will be penalized.